Efficiëntie van raymarching in renderen

Taeke Roukema

Oktober 2022

Samenvatting

Inhoudsopgave

1 Voorwoord

2 Inleiding

2.1 Introductie onderwerp

Renderen is overal. Als je je telefoon opent zie je allerlei gerenderde vormen. Bij het ontbijt zijn verpakkingen volgeprint met teksten die met de computer getekend zijn. Als je langs een bouwterrein loopt zie je hyperrealistische visualisaties van de architectuur. Moderne blockbuster-films zitten tegenwoordig bomvol CGI[1]. En er zijn al tientallen jaren films te zien die helemaal door de computer gemaakt zijn.

Voor Toy Story 3 (Figuur 2.1) werd er gemiddeld zeven uur over gedaan om een frame te renderen [Lehrer, 2010]. En dat terwijl er gebruik werd gemaakt van twee gigantische render farms[2]. Het renderen van films kost niet alleen enorm veel tijd, maar ook veel energie. Het is dus belangrijk dat het zo efficiënt mogelijk gebeurd. Er wordt over de hele wereld voortdurend onderzoek gedaan naar manieren om dit proces efficiënter te maken en te verbeteren. De opkomst van kunstmatige intelligentie begint al bewegingen te maken in de wereld van CGI [Anderson, 2021]. Maar er wordt ook voortdurend voortuitgang gemaakt op fundamentelere manieren. Zo zijn er de afgelopen vijf jaar GPU's[3] van Nvidia op de markt gekomen met ingebouwde support voor realtime raytracing[Alwani, 2018]. Door op het hardware niveau de chips zo te ontwerpen dat ze heel goed zijn in bepaalde berekeningen die gebruikt worden voor het simuleren van licht kunnen GPU's gebruikt worden om voormalig minutendurende processen meer dan zestig keer per seconde uit te voeren.

Figuur 2.1: Een frame uit Toy Story 3, aan de linkerkant worden geen lichtberekeningen gedaan, en aan de rechterkant wel.

Er zijn twee belangrijke maatstaffen waarmee we de efficiëntie van een renderalgoritme kunnen meten. De eerste is vanzelfsprekend: snelheid. Als een frame sneller gerenderd is wordt er minder energie gebruikt en zijn we goedkoper uit.

[1] **C**omputer **G**enerated **I**magery

[2] Een computercluster speciaal gemaakt voor het renderen van CGI, de term was geïntroduceerd in de productie voor Bored Room[Clay, 1990]

[3] **G**raphical **P**rogramming **U**nit

Figuur 2.2: De videogame Minecraft kan gebruik maken van Nvidia GPU's om realtime lichtsimulaties te berekenen.

Maar ook geheugenbezetting is belangrijk om rekening mee te houden. Het geheugen is simpelweg de plaats in de computer waar alle informatie wordt opgeslagen. Als je berekeningen doet moet je ergens de resultaten tussendoor opslaan. Complexe scènes kunnen enorm veel details hebben, die allemaal in het geheugen opgeslagen zijn. Het is niet gratis om extra geheugen toe te voegen, het is dus belangrijk om de geheugenbezetting te minimaliseren.

2.2 Relevantie

Vrijwel alles is tegenwoordig op een manier gerenderd. Objecten zijn gedesigned met gebruik van CAD[4]. Besturingssytemen runnen op een grafische shell. En een meerendeel van advertenties gebruikt CGI.

Volgens Peter Collinridge[Sci, 2016] gebruikt de render farm van pixar 24000 processor cores verdeeld over 2000 computers. Er wordt dus waarschijnlijk gebruik gemaakt van computers met $24000/2000 = 12$ cores. De Ryzen 9 5900x is een voorbeeld van een processor met 12 cores, het energiegebruik zal niet exact hetzelfde zijn maar het ligt bij elkaar in de buurt. De 5900x gebruikt 105 Watt. $105W \cdot 2000 \approx 2,10 \cdot 10^5 W$. Volgens dezelfde bron kostte het renderen van Monster's University twee jaar. Dat is $2 \cdot 365 \cdot 24 \approx 17520h$.

$$210kW \cdot 17520h \approx 3679200kWh$$

Een gemiddeld huishouden in Nederland gebruikt 2479 kWh per jaar. Dit betekent dat het renderen van Monster's University $3679200/2479 \approx 1484$ huishoudens een jaar lang van energie had kunnen voorzien. En dat is alleen nog maar de processorkracht, er gaat ook nog energie naar de moederborden, het geheugen, de koeling en de harde schijven. Kortom, er valt een hoop te besparen.

[4] **C**omputer **A**ided **D**esign

2.3 Onderzoeksvraag/deelvragen

2.3.1 Hoofdvraag WIP

In welke situaties is raymarching een efficiëntere rendertechniek dan polygonaal renderen?

2.3.2 Deelvragen WIP

- Hoe beschrijf je een driedimensionale vorm?
- Hoe werkt raytracing?
- Hoe werkt raymarching?
- Hoe werkt rasterization?
- Hoe beschrijf je een vorm zodat het gerenderd kan worden met raymarching?
- Hoeveel geheugen neemt polygonaal renderen in?
- Hoeveel geheugen neemt renderen met raymarching in?
- Hoe snel is renderen met raymarching vergeleken met polygonaal renderen?
- Hoe kan je met raymarching objecten modelleren?

3 Theorie

3.1 Wat is renderen?

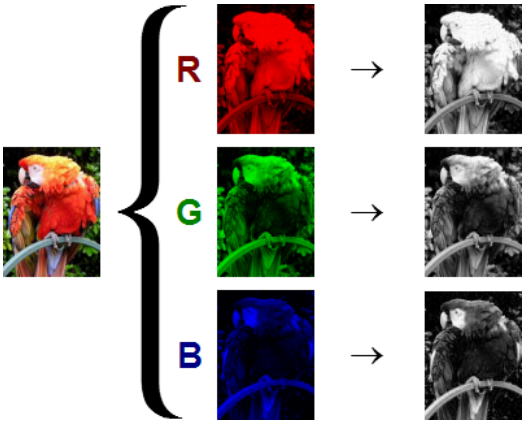
Renderen is, in feite, het weergeven van een representatie van een concept op een beeldscherm. Wij zijn voortdurend bezig met het interacteren met computers, en die interactie verloopt via het beeldscherm. Maar de computer kan uit zichzelf niet zomaar alles tekenen. Daar worden allemaal algoritmes voor geschreven. Een voorbeeld van zo'n algoritme is het tekenen van een rechthoek. In pseudocode zou je dat als volgt voor kunnen stellen:

```
drawRectangle(x1, x2, y1, y2) {
    for (x = x1; x < x2; x++) {
      for (y = y1; y < y2; y++) {
          drawPixel(x, y);
       }
    }
}
```

Het algoritme beschouwt elke pixel die binnen de rechthoek valt en kleurt die pixel. In dit geval wordt dat gedaan door twee loops, die samen alle mogelijke combinaties van x- en y-coördinaten doorlopen.

Renderen omvat, in principe, niks anders dan het aansturen van individuele pixels. Zo'n pixel heeft op de meeste moderne beeldschermen drie waarden die de kleur aansturen: R, G en B, die respectievelijk staan voor rood, groen en blauw. Ze kunnen een geheel getal tussen de 0 en 255 aannemen wat resulteert in 224 mogelijke kleuren.

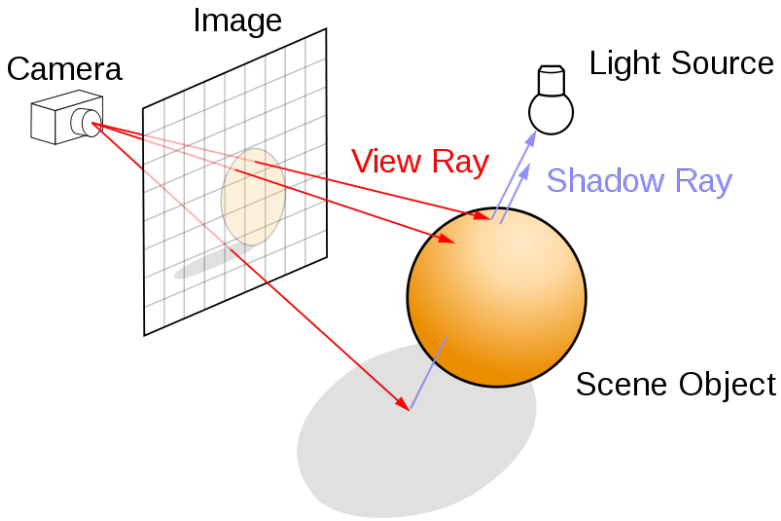
Renderen doen we op een twee-dimensionaal beeldscherm. Dat betekent dat de positie van elke pixel te beschrijven is met twee waarden. Maar de wereld om ons heen kent niet twee, maar drie ruimtelijke dimensies. Door licht dat op ons netvlies valt na weerkaatst te zijn door verschillende objecten kunnen wij die wereld representeren in onze hersenen op een tweedimensionale manier. Camera's gebruiken een gelijksoortige techniek, de lens neemt het licht en projecteert het op een sensor die de intensiteit en de kleur waarneemt. Met het renderen van driedimensionale objecten proberen we deze processen na te bootsen.



Figuur 3.1: De kleuren in een foto kunnen opgesplitst worden in rode, groene en blauwe kanalen.

3.2 Wat is raytracing?

Raytracing zou gezien kunnen worden als de meest voor de hand liggende rendermethode. Het ligt het dichtst in de buurt van het simuleren van echt licht. Het belangrijkste verschil tussen raytracen en licht in onze fysieke wereld is dat we met raytracen alleen het licht beschouwen wat zichtbaar is voor ons perspectief. Om dit te bereiken voeren we de lichtstralen niet af vanuit de lichtbron, maar vanuit de camera, vervolgens kaatsen we de straal af naar de lichtbron om te kijken hoe vel die plek zou zijn, en of er een ander object in de weg zit die een schaduw zou kunnen werpen. Op Figuur 3.2 is zichtbaar hoe dat raytracen in werking gaat.



Figuur 3.2: Een diagram die laat zien hoe raytracen werkt

3.3 Wat zijn polygonen?

3.4 Wat is rasterization?

In de vroege jaren van de ontwikkeling van de computer was de rekenkracht miniscuul vergeleken met waar we vandaag de dag toegang tot hebben. Gordon Moore, mede-oprichter van Intel en legendarische informaticus, stelde in 1965 Moore's Law voor (al had het toen nog een andere naam). Moore's Law stelt dat elke twee jaar het aantal transistors in een *integrated circuit* verdubbelt [Moore, 1965]. Dit betekent dus dat 40 jaar geleden de computers $\frac{1}{2^{20}}$ keer zoveel rekenkracht hadden. Wat ongeveer één miljoenste is. Het was toen simpelweg niet mogelijk om algoritmes zoals raytracing toe te passen, omdat de complexiteit van dat soort algoritmes de capaciteit van de computers ver te boven gingen.

De oplossing was relatief simpel, in plaats van voor elke pixel te checken of er een object in de weg zit kijk je alleen maar naar de positie van de hoeken van een vorm. Als je daar vervolgens de schermcoördinaten van hebt kan je de vorm gewoon invullen, wat een vrij goedkoop proces is.

3.5 Wat is raymarching?

3.6 Hoe werkt het geheugen?

4 Hypothese

5 Ontwikkeling

5.1 Hardware

5.2 Software

5.2.1 Besturingssysteem

Het uitvoeren en programmeren van de code zal volledig met Linux gedaan worden. Deze keuze is om meerdere redenen gemaakt. Ten eerste is Linux de standaardkeuze voor developers om te ontwikkelen. Windows en MacOS zijn ontwikkeld als product voor de gebruiker, terwijl Linux ontwikkeld is voor stabiliteit en betrouwbaarheid. Dit heeft als effect dat er met Linux veel minder tegen het besturingssysteem in gewerkt hoeft te worden. Bovendien zijn alle tools die gebruikt worden voor dit onderzoek FOSS[5], en daardoor voor zowel Windows als Linux beschikbaar. Dus qua support maakt het geen verschil.

Als distributie[6] is voor Manjaro[7] gekozen. Dit is een fork van Arch Linux. Arch Linux staat erom bekend dat het altijd de nieuwste versie van software support. Dit komt doordat het een *rolling release model* heeft, in tegenstelling tot *fixed release*, waar distributies zoals Debian gebruik van maken. Bovendien heeft Arch Linux toegang tot de Arch User Repository (AUR). Dat is een enorme collectie van software die gebruikers zelf kunnen uploaden naar de Arch servers, met helpers zoals *yay* kan je met één commando vrijwel alle software die beschikbaar is op GNU/Linux op de computer installeren. De combinatie van deze voordelen maken Arch Linux een erg aantrekkelijke distributie om te gebruiken als programmeur. Manjaro voegt de rest van de functies toe aan het besturingssysteem, zoals een Desktop Environment (DE) en een terminal.

5.2.2 Programmeertaal

Er zijn talloze programmmeertalen die geschikt zijn voor graphics programming. Daarom was de keuze voor de programmeertaal niet makkelijk. Zelf heb ik al jarenlang ervaring met Python[8], maar deze taal staat niet bekend om de snelheid. Dit komt doordat het een *interpreted* taal is. Dat betekent dat de code live gelezen wordt wanneer gerund. Dit staat tegenover *compiled* talen, die de code eerst compileren naar machinetaal. Die machinetaal is veel efficiënter te lezen door computers, waardoor de snelheid toeneemt. Een andere optie was Javascript, het grote

[5] **F**ree and **O**pen **S**ource **S**oftware

[6] Linux zelf is slechts een *kernel* die de interactie tussen de hardware en de software regelt. Bovenop deze *kernel* bestaan distributies die het een werkend besturingssysteem maken.

[7] https://manjaro.org/

[8] https://www.python.org/

voordeel van deze taal is dat hij speciaal voor het web gemaakt is. Hierdoor zou het delen van het gemaakte project met anderen zo simpel zijn als het doorsturen van een url. Bovendien maakt Javascript op moderne browsers gebruik van Just In Time (JIT) compilation. Dat is een combinatie tussen *interpreted* en *compiled* waar de code live omgezet wordt in machinetaal voordat het gerund wordt. Maar toch is zelfs Javascript niet snel genoeg. Bovendien missen beide talen iets wat erg belangrijk is in computer graphics: controle over het geheugen. Scènes kunnen enorm complex zijn dus het is belangrijk dat die zo efficiënt mogelijk in het geheugen geplaatst worden, en het geheugen moet weer gewist worden wanneer het niet meer gebruikt wordt. Python en Javascript geven allebei niet die controle, in plaats daarvan probeert de *interpreter* zelf zo efficiënt mogelijk het geheugen te gebruiken. Om deze redenen heb ik gekozen voor C++, deze taal is in 1985 uitgevonden door Brane Stroustrup en wordt vandaag de dag nog door 20,17% van Stack Overflow gebruikers gebruikt[Sta, 2022]. De taal is ontwikkeld als extensie voor C, waardoor het moderne functies heeft zoals *Object Oriented Programming* (OOP) en datastructuren. Maar het heeft tegelijkertijd alle voordelen die C heeft als low-level taal.

5.2.3 Framework

C++ heeft uit zichzelf nog geen grafische capabiliteiten. Daar is een framework voor nodig. Moderne grafische kaarten zijn allemaal gemaakt met speciale specificaties, die ervoor zorgen dat het besturingssysteem weet hoe hij moet communiceren met de GPU. Er zijn verschillende van deze specificaties met verschillende doelen. Zo heb je DirectX, die specifiek gemaakt is voor Windows. En wat algemenere API[9] is OpenGL (Open Graphics Library). Met C++ is het dan ook mogelijk om direct gebruik te maken van deze API, net als in de meeste programmeertalen.

Maar toch heb ik daar niet voor gekozen. Dit is omdat OpenGL heel goed is in het implementeren van bestaande rendermethodes, waar de GPU ook voor ontwikkeld is. Dit maakt het ideaal voor het bouwen van videogames, omdat het daar heel snel in is. Maar minder voor dit specifieke onderzoek. Ik wil objectief vergelijken hoe de verschillende rendermethoden tegen elkaar opwegen, als de gebruikelijkere methodes heel goed geoptimaliseerd zijn door de GPU en OpenGL zou dat oneerlijk zijn en de data onbetrouwbaar maken. Daarom heb ik gekozen voor raylib[10]. Raylib is een zeer minimalistische *library* die alle basistools geven die we nodig hebben om te kunnen tekenen op een canvas, terwijl het tegelijkertijd razendsnel blijft.

[9] **A**pplication **P**rogramming **I**nterface
[10] https://www.raylib.com/

5.2.4 Integrated Development Environment

Ik ga al mijn programmeren doen in Visual Studio Code[11] omdat het een mooie simpele text editor is die precies doet wat ik wil. Het geeft goede IntelliSense[12], syntax highlighting en het geeft een goed overzicht van het project. Bovendien heeft het een enorme markt van plugins die het product nog meer verbeteren. Zo gebruik ik de Vim keybinds plugin om de efficiënte workflow van de editor Vim[13] te emuleren.

5.3 Programmeren

5.3.1 Specificaties

Voor het onderzoek ga ik twee verschillende renderers ontwikkelen. De eerste gebruikt raytracing, en de objecten worden beschreven met raymarching. De tweede gebruikt ook raytracing alleen worden de objecten polygonaal beschreven. Het is voor het onderzoek belangrijk dat de twee algoritmes zo veel mogelijk op elkaar lijken, zodat het vergelijkende onderzoek iets objectiefs kan zeggen over de effectiviteit van de verschillende technieken. Vandaar dat hier eerst de specificaties worden beschreven waar de algoritmes aan zullen moeten doen. Deze specificaties kunnen uitgebreid worden of ingekort gebaseerd op de progressie over de tijd.

De renderer moet kunnen renderen vanuit een **camera** die een bepaalde positie en draaiing heeft binnen de driedimensionale ruimte. De camera heeft ook een *field of view* (fov), die aangeeft hoe breed het zichtveld is. Uit de camera komt uit elke pixel een *ray*. De resolutie van het beeldscherm wordt in het programma aangegeven. Waarschijnlijk willen we renderen op een resolutie van 500×500. De *rays* hebben ook een positie en een rotatie. De *ray* zal een methode hebben die de plaats van raaking met een object teruggeeft. De objecten zijn uiteraard anders beschreven in de twee verschillende programma's, maar ze hebben wel delen gemeen. Zo heeft een object altijd een positie, een draaiing en een grootte. Verder zal elk object materiaal-eigenschappen hebben. Bovendien willen we een aantal primitieve vormen hebben die standaard beschikbaar zijn. Voor raymarching zal dit triviaal zijn aangezien elke vorm slechts een *signed distance function* is. Maar voor het polygonale renderen moeten hier nog aparte algoritmes voor komen.

De eerste generatie van de algoritmes zal gebruik maken van de diepteinformatie om de kleur te bepalen. Hoe verder hoe donkerder. Dit zal niet voor mooie resultaten zorgen maar het is een goede manier om de werking van de verschillende

[11] https://code.visualstudio.com/

[12] Verzamelnaam voor tools die helpen in het schrijven van code zoals: code completion en informatie over parameters

[13] https://www.vim.org/

technieken te testen. In deze fase zijn er nog geen lichten nodig.

Hierna, in de tweede generatie, introduceren we lichten. Als een *ray* het object raakt kaatsen we het af naar het licht. We testen of het een ander object raakt om te kijken of er een schaduw getekend moet worden. Verder meten we de afstand van het raakpunt tot de lichten om te bepalen hoe belicht dat punt is. Om de intensiteit te bepalen van een lichtstraal gebruiken we de omgekeerde kwadratenwet.

5.3.2 Raylib uitproberen

5.3.3 Header Hell

5.3.4 Rotatie? Qu'est-que c'est?

6 Methode

6.1 Variabelen

6.2 Meetmethoden

7 Resultaten

7.1 Snelheid

7.2 Geheugenbezetting

7.3 Renders

8 Nauwkeurigheidsanalyse

9 Conclusie

10 Discussie

11 Nawoord

Bedankt aan mijn moeder Arria Gosman.

12 Literatuurlijst

Referenties

[Sci, 2016] (2016). Rendering.

[Sta, 2022] (2022). Stack overflow 2022 developer survey.

[Alwani, 2018] Alwani, R. (2018). Microsoft and nvidia tech to bring photorealistic games with ray tracing.

[Anderson, 2021] Anderson, M. (2021). Nerf moves another step closer to replacing cgi.

[Clay, 1990] Clay, J. (1990). Making of bored room (production).

[Lehrer, 2010] Lehrer, J. (2010). How toy story 3 was made.

[Moore, 1965] Moore, G. (1965). Cramming more components onto integrated circuits.

13 Logboek

Activiteit	Datum	Tijd (m)	Totale tijd (h)	% Voltooid
Programmeren	20220906	45	0.8	0.94%
Programmeren	20220908	30	1.3	1.56%
Gesprek met begeleider	20220909	20	1.6	1.98%
Programmeren	20220921	90	3.1	3.85%
Inhoudsopgave Opzet	20220928	35	3.7	4.58%
Schrijven Theorie/Achtergrond Renderen	20220930	45	4.4	5.52%
Opzetten LaTeX Document	20221002	120	6.4	8.02%